1. [3 marks] Write a C++ function that finds the largest integer less than or equal to the square root of *n* for a given `unsigned long n`. You may not use any floating-point numbers or calculations.

```
unsigned long sqrt( unsigned long n );
```

2. [3 marks] Write a C++ function that calculates and returns the following sum:

$$(1) + (1 + 4) + (1 + 4 + 9) + (1 + 4 + 9 + 16) + \cdots + (1 + 4 + 9 + \cdots + n^2)$$

where *n* is an `unsigned int` parameter of the function. If *n* is zero, the result is also zero.

```
unsigned int square_sum( unsigned int n );
```

3. [6 marks] Write a C++ function that "rotates" an `int` array to the right, by moving the first entry of the array to the second, the second entry to the third, and so on until it moves the last entry to the first position. The function is passed the array and the capacity of the array, and the specified operation is performed on that array. Give the function and parameters appropriate names and determine the return type, if necessary.

4. [10 marks] Write a C++ function that takes two `unsigned short` as parameters and interleaves the bits into an `unsigned int` that is returned. For example, the two `short` integers `m` and `n`

```
m: 0000000000000001     n: 1111111111111110
```

would return the `unsigned int`

```
01010101010101010101010101010110
```

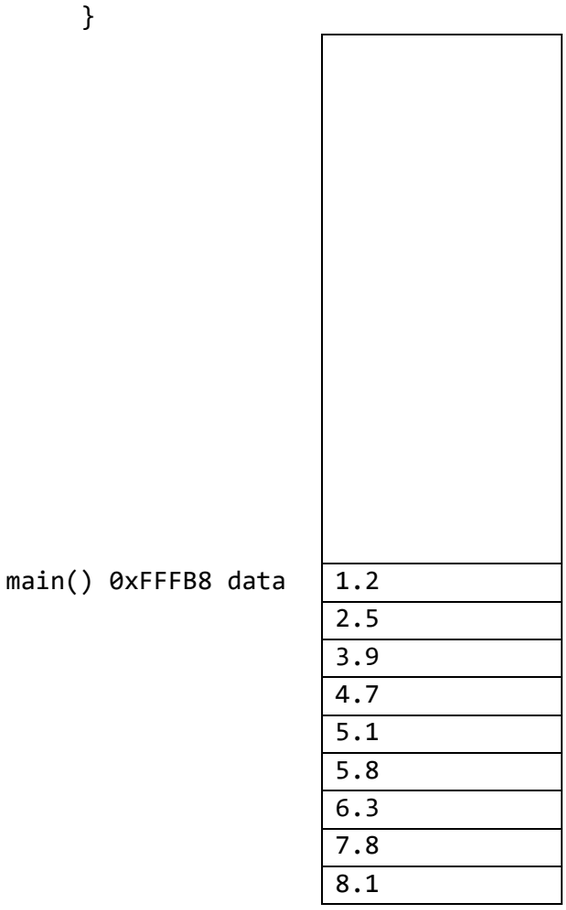
That is, the even bits (note that we start counting with zero (which is even) with the least-significant (the right-most) bit) are the bits from `n` and the odd bits are the bits from `m`.

```
unsigned int mingle( unsigned short m, unsigned short n );
```

5. [7 marks] The following is an implementation of a binary search together with a main function that calls it. Assume all library inclusions and function declarations are elsewhere. Draw the state of the call stack just before a return statement marked with *** is executed. Your call stack should include local variables and parameter values. You will need the address of the array, but you don't have to specify the addresses of other local variables and parameters. Be sure to distinguish separate function calls on the stack.

```
// Search array entries from index m up to and including index n
bool binary_search(
    double target, double array[], std::size_t m, std::size_t n
) {
    if ( m > n ) {
        return false;  // ***
    } else if ( n - m <= 1 ) {
        return (array[m] == target ) || (array[n] == target); // ***
    } else {
        std::size_t mid{ (m + n)/2 };
        if ( array[mid] == target ) {
            return true; // ***
        } else if ( target < array[mid] ) {
            return binary_search( target, array, m, mid - 1 );
        } else {
            return binary_search( target, array, mid + 1, n );
        }
    }
}

int main() {
    double data[9]{ 1.2, 2.5, 3.9, 4.7, 5.1, 5.8, 6.3, 7.8, 8.1 };
    binary_search( 1.8, data, 0, 8 );
    return 0;
}
```



6. [10 marks] In the course, and in the previous question, you have seen the implementation of a binary search. Instead, here you will implement a ternary search, as follows.

The function

```
bool ternary( int val, int arr[], std::size_t m, std::size_t n );
```

will search for value `val` in the sorted array `arr` from entry `m` to entry `n`, inclusive. You may assume the array is sorted and that `m <= n`.

- If there are seven or fewer entries to search, then just do a linear search: check the entries from `arr[m]` to `arr[n]` and check if any of these equals to `val` – return true if one equals to `val` and false otherwise.
- If there are eight or more entries to be searched, choose an entry close to one third of the interval, and choose an entry close to the second third of the interval.
    - If either of these are the value you are searching for, return true.
    - Otherwise, recursively call `ternary()` searching the first third, second third, or third third, as appropriate.

To search for a value in an array with capacity `cap`, the first call to `ternary()` is with `m = 0` and `n = cap - 1`, which searches the entire array.
For example, if m = 15 and n = 34, then we could check entries 21 and 27, and if neither of these entries equaled `val`, then we would recursively call `ternary()` with the arguments (15, 20), (22, 26), or (28, 34), as appropriate. We don't require you to get exactly 21 or 27 in this case, but your calculation should at least approximately divide the interval `m` to `n` into three approximately equal sub-intervals.

7. Given this class definition of a matrix class, implement the member functions:

```
class Matrix {
  public:
    Matrix( unsigned int m, unsigned int n );
    Matrix( Matrix const &orig );

   ~Matrix();

    double &operator()( unsigned int i, unsigned int j );

    Matrix &operator*=( double c );

  private:
    unsigned int m_;
    unsigned int n_;
    double      *array_;
};
```

The m × n matrix is implemented as a one-dimensional array where the first row occupies the first n entries, the second row occupies the next n entries, and so on, so the entry i,j is stored at entry n*i + j. The rows will be indexed from 0 to m – 1 and the columns will be indexed from 0 to n – 1. Thus, if the matrix is 5 × 4, the indices will go from (0, 0) to (4, 3).

1. [2 marks] The constructor takes the two dimensions *m* and *n*, allocates an array of sufficient capacity to store all *m* × *n* entries of the array, and sets all the entries to zero.
   `Matrix::Matrix( unsigned int m, unsigned int n )`

2. [3 marks] The copy constructor makes a copy of parameter `orig`. This must have a newly allocated array that contains a copy of the entries of the array associated with the original matrix.
   `Matrix::Matrix( Matrix const &orig ) {`

3. [2 marks] The destructor deletes any memory allocated in the constructor and ensures there are no dangling pointers.
   `Matrix::~Matrix() {`

4. [1 mark] If x is an instance of this matrix class, we will access the entries by overloading the function operator so that x(3, 4) allows you to either access or assign to entry (3, 4) of the matrix. Thus, this function will return the appropriate entry in `array_[`*k*`]` where *k* corresponds to where the entry is stored in the array.
   `double &Matrix::operator()( unsigned int i, unsigned int j ) {`

5. [2 marks] The automatic multiplication operator *= multiplies each entry in the matrix by the parameter `c`, returning this matrix.
   `Matrix &Matrix::operator*=( double c ) {`

8. [8 marks] In lectures, you saw an array class

```
class Array {
   public:
     Array( std::size_t cap );
     Array( Array const &orig );
    ~Array();
     void interpolate();
     // Other member functions...

   private:
     std::size_t cap_;
     double *array_;
 };
```

Implement a new member function `void interpolate()` that replaces the array of the current capacity cap_ with a new array that has capacity `2 * cap_ - 1` where:

1. The entries in the original array are copied to each alternate entry of the new array starting with index 0.
2. The intermediate entries are assigned the average of the two entries surrounding it.

If the capacity of the array is 0 or 1, `interpolate()` does nothing.
As an example, if the entries of the array were

`0.5, 0.9, 0.7, 1.2, 1.5, 1.6`

the new array would contain the entries

`0.5, 0.7, 0.9, 0.8, 0.7, 0.95, 1.2, 1.35, 1.5, 1.55, 1.6`

```
void Array::interpolate() {
```

9. [8 marks] You have a linked list that stores double-precision floating-point numbers:

```
class Node {
  public:
    double value() const;
    Node *get_next() const;
    void set_next( Node *p_new_next );
  private:
    double value_;
    Node  *p_next_;
};
```

where the member functions work as expected from the description in the lectures.

Write a member function `make_sorted()` for the linked-list class

```
class Linked_list {
  public:
    void make_sorted();
    // ...
  private:
    std::size_t count_;
    Node       *p_head_;
};
```

that walks through the linked list and deletes any nodes that make the list not sorted in ascending order (the value stored in the next node is greater than or equal to the value stored in the current node). Make sure to not leak memory!

As an example, if the nodes in the linked list contain the values

p_head_→1.1→3.9→6.9→7.7→7.3→4.0→9.4→2.1→7.5→4.5→7.4→9.9→6.2→`nullptr`

then after the call to this function, it would contain

p_head_→1.1→3.9→6.9→7.7→9.4→9.9→`nullptr`

```
void Linked_list::make_sorted();
```

10. [6 marks] You have a special linked list where there is no need to remove nodes from the linked list, so there is no pop function. This special linked list has the member variable `p_head_` that stores the address of the first node, and each node has a `p_next_` that stores the address of the next node. The twist, however, is that the last node in the linked list stores, instead of `nullptr`, the address of the first node. This is called a circular linked list. Write a destructor for this class.

```
List::~List() {
  // You may access p_head_ and for each node you may access p_next_.
  // You should call delete on each node in the linked list exactly once.
```

11. [3 marks] You are given the following C++ programs. Answer the questions at the bottom of the page.

```
#include <iostream>

class A;
class B;
class C;
class D;
```

LEFT	RIGHT

class A {
  public:
  void print() const;
};
 	
void A::print() const {
  std::cout << "A";
}
 	
class B : public A {
  public:
  void print() const;
};
 	
void B::print() const {
  std::cout << "B";
}
 	
class C : public A {
  public:
  void print() const;
};
 	
void C::print() const {
  std::cout << "C";
}
 	
class D : public B {
};
 	
	class A {
	  public:
	  virtual void print() const;
	};
	
	void A::print() const {
	  std::cout << "A";
	}
	
	class B : public A {
	  public:
	  void print() const **override**;
	};
	
	void B::print() const {
	  std::cout << "B";
	}
	
	class C : public A {
	  public:
	  void print() const **override**;
	};
	
	void C::print() const {
	  std::cout << "C";
	}
	
	class D : public B {
	};

```
int main() {
  A a{};
  B b{};
  C c{};
  D d{};

  A *data[4]{ &a, &b, &c, &d };

  for ( std::size_t k{ 0 }; k < 4; ++k ) {
    data[k]->print(); // Line 1
  }
  return 0;
}
```

What is the output if you use the LEFT code?

What is the output if you use the RIGHT code?

With the RIGHT code, what object-oriented programming concept does Line 1 exercise?

12. [13 marks] The class `Poly` allows the user to declare a polynomial of a given degree. One constructor has the following declaration

```
Poly( double array[], unsigned int degree );
```

This constructor takes an array with degree + 1 entries, and constructs an instance of a polynomial. The entry array[k] is the coefficient of x^k. You will simply use this class.

You will write a function `regress(…)` that takes a pair of arrays and the capacity of both arrays (which have the same capacities), and that returns an instance of a `Polynomial`:

```
Poly regress( double xs[], double ys[], std::size_t n );
```

It determines the polynomial as follows:

1. If the capacity n is zero, return the zero polynomial (a polynomial of degree zero where the constant is zero).
2. Otherwise, if all the *x*-values `xs` are the same, then return a polynomial of degree zero where that constant is the average of the *y*-values `ys`.
3. Otherwise, return the polynomial of degree 1 with linear coefficient a and constant b where:

$$a = \frac{SP_{xy} \times n - S_x \times S_y}{SS_x \times n - (S_x)^2} \qquad \text{and} \qquad b = \frac{SS_x \times S_y - SP_{xy} \times S_x}{SS_x \times n - (S_x)^2},$$

where S_x is the sum of the *x*-values, S_y is the sum of the *y*-values, SS_x is the sum of the squares of the *x*-values (so $x_0^2 + \cdots + x_{n-1}^2$) and SP_{xy} is the sum of the products of the *x*- and corresponding *y*-values, so (so $x_0y_0 + \cdots + x_{n-1}y_{n-1}$). You are required to calculate these various sums using helper functions. Note that some marks will be assigned for good design.